Bloque 3 · Tema 4

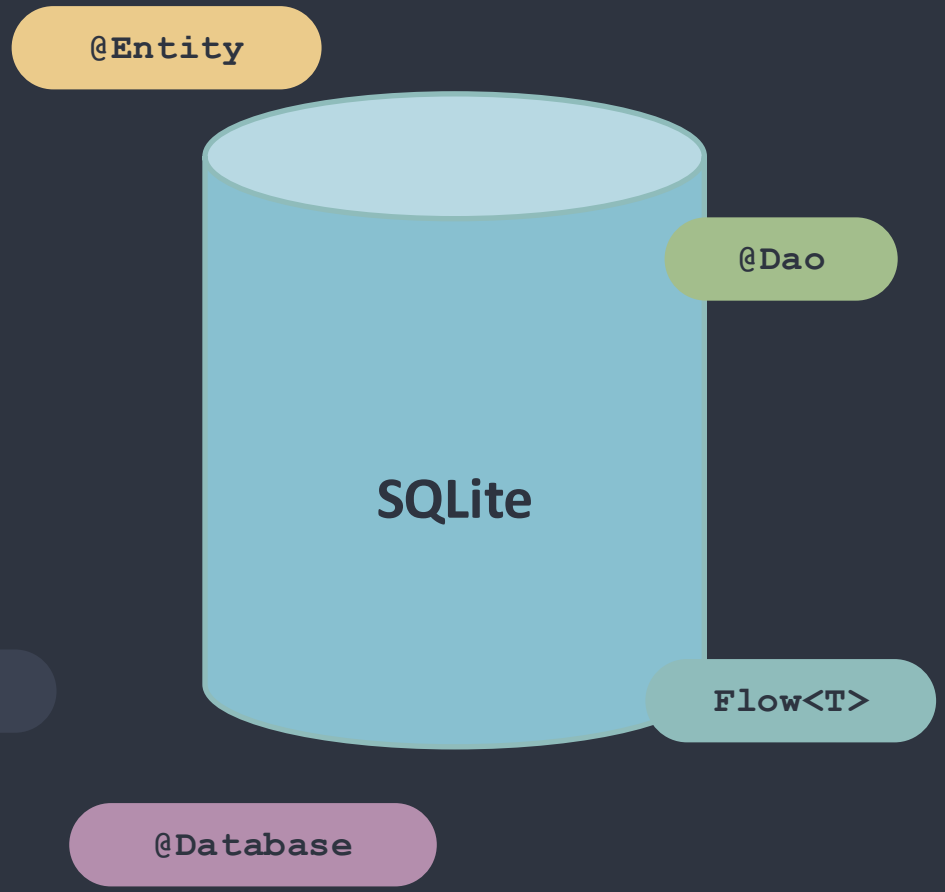
ROOM

Persistencia local en Android con Kotlin y Jetpack Compose

6 horas

AppDummy · biblioteca personal

RA2 · RA3



Hoja de ruta del tema

9 apartados

1 **¿Qué es Room?**
Abstracción sobre SQLite

2 **@Entity**
Una clase = una tabla

3 **@Dao**
Flow, suspend y @Transaction

4 **@Database**
Punto de entrada y singleton

5 **TypeConverters**
Tipos que SQLite no conoce

6 **Relaciones 1:N**
@Relation y @Embedded

7 **Integración MVVM**
Repository, Flow frío/caliente y stateIn

8 **AppContainer**
Dependencias sin framework de DI

9 **Testing**
Base de datos en memoria

¿Qué vamos a evaluar?

RA2 · RA3

RA2

Desarrolla aplicaciones para dispositivos móviles analizando y empleando las tecnologías y librerías específicas.

RA3

Desarrolla programas que integran contenidos multimedia analizando y empleando las tecnologías y librerías específicas.

RA2-a	Estructura de clases de la app
RA2-b	Ventanas, menús, alertas y controles (apps gráficas)
RA2-f	Conexión con almacenes de datos y persistencia
RA2-g	Pruebas de interacción en emuladores
RA2-i	Documentar el proceso de desarrollo
RA3-c	Conversión de datos multimedia
RA3-d	Procesamiento de datos
RA3-e	Eventos, tipos y excepciones
RA3-h	Depurar y documentar

★ *RA2-f es el criterio central de este tema*

01

Fundamentos

Qué es Room, sus componentes y cómo se añade al proyecto



Room: una capa de abstracción sobre SQLite

Jetpack

Tu código Kotlin

Entidades, DAOs, ViewModels…

Room

Anotaciones + código generado por KSP

SQLite

Motor de base de datos relacional

Almacenamiento interno

Archivo appdummy_database

Genera el SQL por ti

A partir de anotaciones en clases Kotlin.



Errores en compilación

Las consultas SQL se verifican antes de ejecutar.



Reactivo de serie

Integración nativa con corrutinas y Flow.



Esquema evolutivo

Facilita las migraciones de versión.

Sin Room frente a con Room

SQLite puro

```
val db = openOrCreateDatabase(...)
val cursor = db.rawQuery(
    "SELECT * FROM libros", null)
val libros = mutableListOf<Libro>()
while (cursor.moveToNext()) {
    libros.add(...)    // mapeo manual
}
cursor.close()         // ¡no olvidarlo!
```

Room

```
@Entity data class Libro(...)

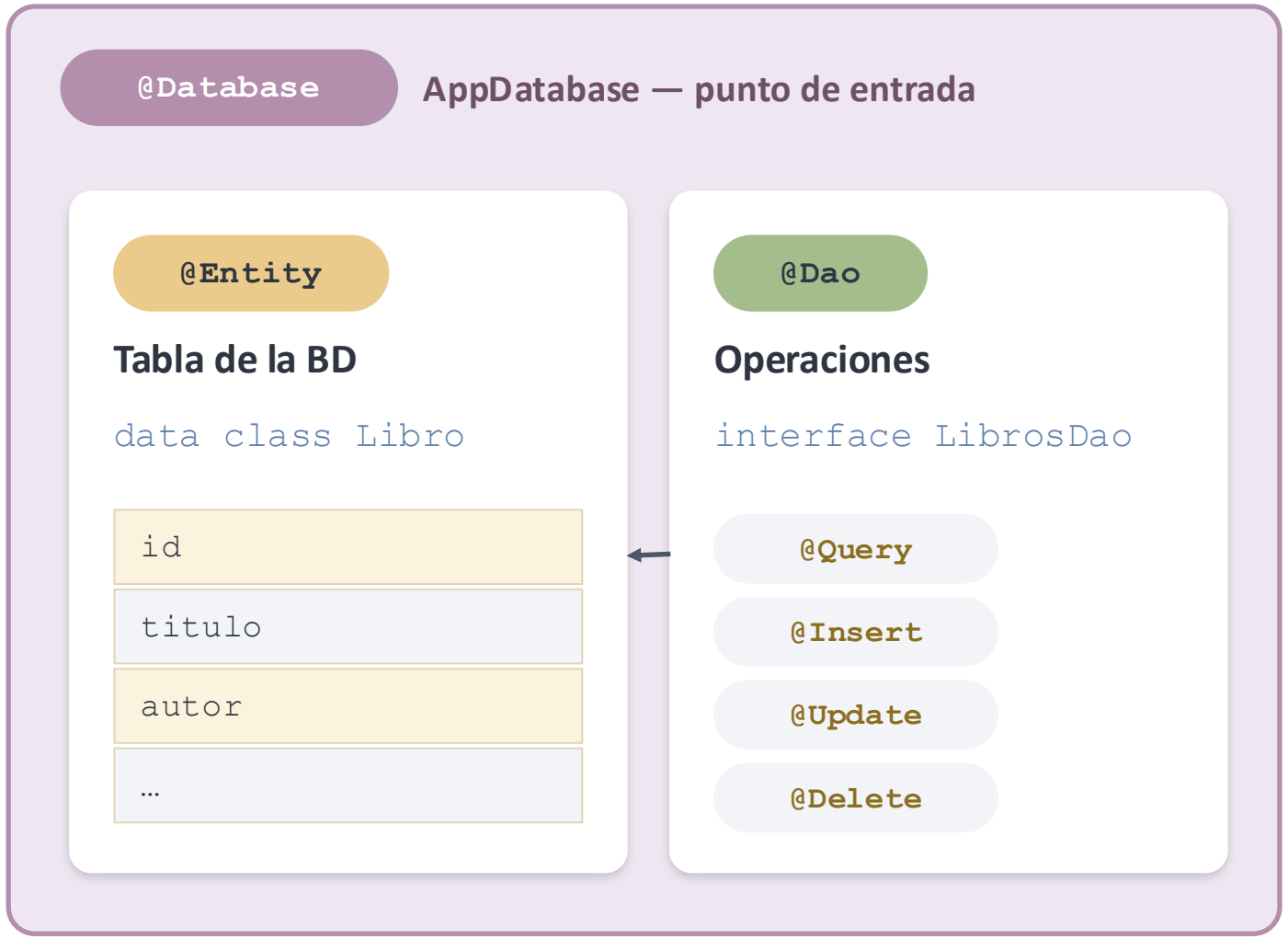
@Dao interface LibrosDao {
    @Query("SELECT * FROM libros")
    fun observarTodos(): Flow<List<Libro>>
}

@Database(...)
abstract class AppDatabase
// Room genera todo lo demás
```

Errores SQL solo en tiempo de ejecución

Errores SQL detectados al compilar

Los tres componentes de Room



@Entity

Define la estructura de una tabla. Cada propiedad es una columna.

@Dao

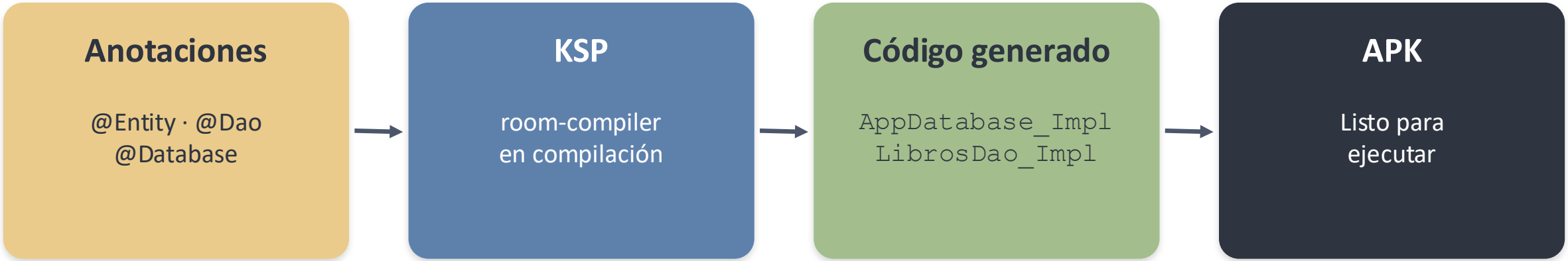
Interfaz con las operaciones. Room genera su implementación.

@Database

Clase abstracta que reúne entidades y DAOs. Una sola instancia.

KSP: el código que no escribes

KSP2



KAPT queda obsoleto

Desde Kotlin 2.0, Room se procesa siempre con KSP. En el bloque de dependencias se usa ksp(...), nunca kapt(...).

KSP2 (≥ 2.3.0)

Su versión ya no está atada a la del compilador de Kotlin: se puede actualizar uno sin el otro. KSP1 queda obsoleto.

Dependencias necesarias

Gradle KTS

```
build.gradle.kts (proyecto)
plugins {
    id("com.google.devtools.ksp")
        version "2.3.10" apply false
    id("androidx.room")
        version "2.8.5" apply false
}
```

`room-runtime`	Núcleo de la biblioteca
`room-ktx`	Extensiones de corrutinas
`room-compiler`	Generador de código (ksp)
`room-testing`	Utilidades para pruebas

```
build.gradle.kts (módulo app)
plugins {
    id("com.google.devtools.ksp")
    id("androidx.room")
}

room {   // exporta el esquema (migraciones)
    schemaDirectory("$projectDir/schemas")
}

dependencies {
    implementation("androidx.room:room-runtime:2.8.5")
    implementation("androidx.room:room-ktx:2.8.5")
    ksp("androidx.room:room-compiler:2.8.5")
    testImplementation(
        "androidx.room:room-testing:2.8.5")
}
```

02

Los tres pilares

@Entity · @Dao · @Database



@Entity: de data class a tabla

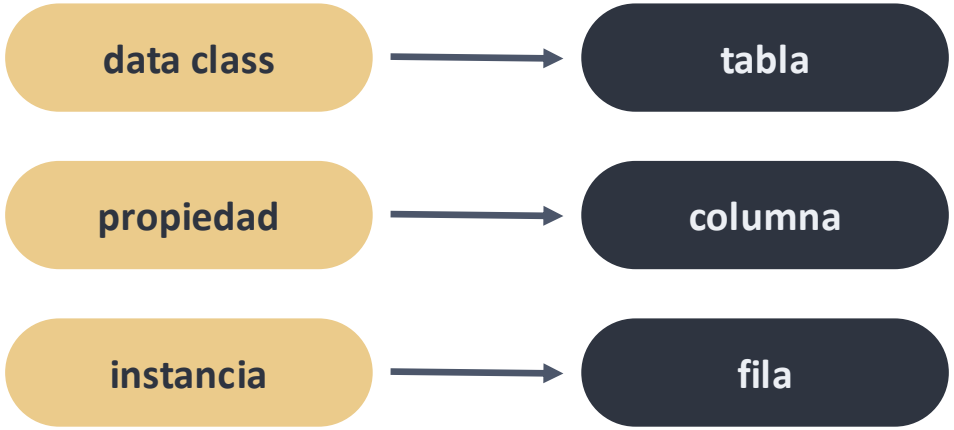
@Entity

data/model/Libro.kt

```kotlin
@Entity(tableName = "libros")
data class Libro(
    @PrimaryKey
    val id: String,
    @ColumnInfo(name = "titulo")
    val titulo: String,
    val autor: String,
    val year: Int? = 1900,
    val isbn: String,
    val cover: String? = null,
    // Solo locales: Gson los deja a false
    // (valor por defecto de Boolean en la JVM)
    @ColumnInfo(name = "es_favorito")
    val esFavorito: Boolean = false,
    val leido: Boolean = false
)
```

Tabla libros

id	titulo	autor	year	es_favorito
OL45883W	Hail Mary	A. Weir	2021	1
OL27482W	Juego de…	G. R. R. M.	1996	0
OL80611W	Festín de…	G. R. R. M.	2005	0



Reglas de oro de @Entity

@Entity

tableName único

Si no se indica, Room usa el nombre de la clase en minúsculas.

Una única @PrimaryKey

autoGenerate = true la asigna Room; en AppDummy el id llega de OpenLibrary.

Tipos complejos

List<String> o Date necesitan un TypeConverter.

Índices

Aceleran las búsquedas frecuentes por una o varias columnas.

Índices en @Entity

```
@Entity(
    tableName = "libros",
    indices = [
        Index(value = ["titulo"]),
        Index(value = ["autor", "isbn"])
    ]
)
data class Libro(...)
```

¿Por qué id: String?

OpenLibrary identifica cada obra con un código alfanumérico, como **`/works/OL45883W`**.

@Dao: la regla fundamental

@Dao

Flow<T> · sin suspend

Consultas de observación

Room vuelve a emitir automáticamente cada vez que cambia la tabla.

```
@Query(SELECT * FROM libros
        ORDER BY year DESC)
fun observarTodos(): Flow<List<Libro>>

@Query(SELECT * FROM libros
        WHERE id = :id)
fun observarPorId(id: String): Flow<Libro?>
```

suspend fun

Escrituras y consultas puntuales

Se ejecutan una vez, fuera del hilo principal, desde una corrutina.

```
@Update
suspend fun actualizar(libro: Libro)

@Delete
suspend fun eliminar(libro: Libro)

@Query("SELECT * FROM libros WHERE id = :id")
suspend fun obtenerPorId(id: String): Libro?
```

Room reactivo: no hay que refrescar la UI

Flow



¿Qué hace cada operación de escritura?

`@Dao`

Operación	Comportamiento	¿Conserva esFavorito?
`@Insert(IGNORE)`	Si el id ya existe, no hace nada y devuelve -1L	✓ No toca la fila
`@Insert(REPLACE)`	Borra la fila existente y la vuelve a insertar	✗ Se pierde
`@Upsert`	INSERT y, si hay conflicto, actualiza todas las columnas	✗ Se sobrescribe
`@Update / @Delete`	Actualiza o elimina la fila por su clave primaria	✓ Lo que traiga el objeto
`@Query("UPDATE …")`	Actualiza solo las columnas indicadas	✓ Sí, si no se incluye

Todas son suspend fun y Room ya las ejecuta dentro de una transacción. @Upsert existe desde Room 2.5.

El problema del favorito perdido

`@Upsert`

En Room (local)

```
id = "3"
titulo = "Festín de
cuervos"
esFavorito = true
```

Llega de la API

```
id = "3"
titulo = "Festín de
cisnes"
esFavorito = false
```



`@Upsert`

```
titulo = "Festín de cisnes"
esFavorito = false
```

Se pierde la preferencia del usuario

`upsertConservandoFavorito`

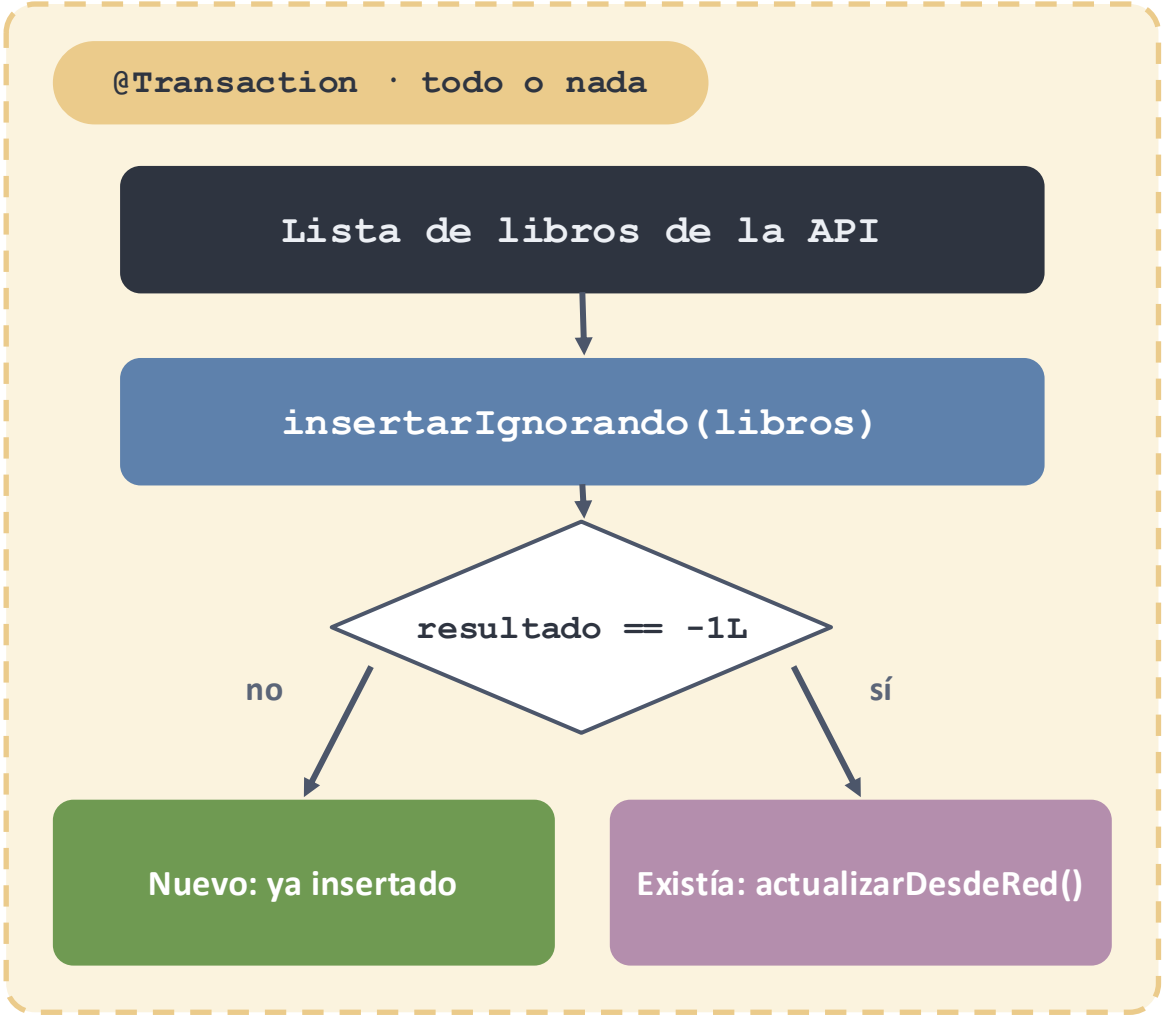
```
titulo = "Festín de cisnes"
esFavorito = true
```

Se actualizan solo los campos de la API

Vídeo del tema: Favorito perdido

La solución: una @Transaction a medida

@Transaction

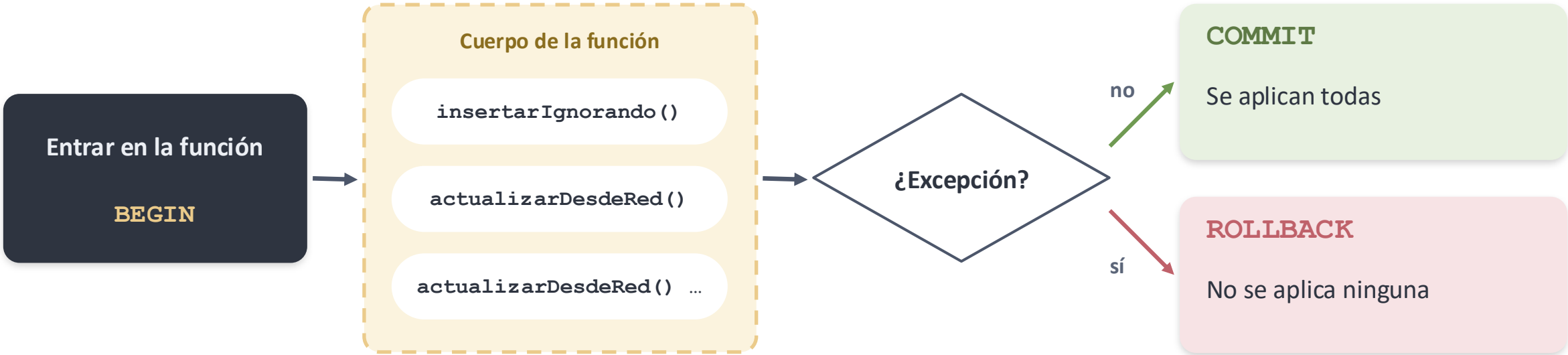


LibrosDao.kt

```kotlin
@Transaction
suspend fun upsertConservandoFavorito(
    libros: List<Libro>
) {
    val resultados = insertarIgnorando(libros)
    libros.forEachIndexed { index, libro ->
        if (resultados[index] == -1) {
            // Solo columnas que vienen de la API
            actualizarDesdeRed(
                id = libro.id,
                titulo = libro.titulo,
                autor = libro.autor,
                year = libro.year,
                isbn = libro.isbn,
                cover = libro.cover)
        }
    }
}
```

¿Qué es una transacción?

`@Transaction`



Unidad indivisible

O se completan todas las operaciones o no se aplica ninguna: la BD nunca queda a medias.

Room la gestiona

La abre al entrar en la función, la confirma si termina bien y la revierte si se lanza una excepción.

Además, más rápido

SQLite confirma los cambios en disco una sola vez, no una vez por operación.

¿Cuándo se utiliza @Transaction?

`@Transaction`

Caso	Dónde se coloca	Qué garantiza
1 **Varias operaciones que deben ir juntas**	Función con cuerpo que llama a otras funciones del DAO	Atomicidad: todas las escrituras se aplican o ninguna
2 **Consultas que devuelven una clase con @Relation**	`Junto a @Query`	Room lanza varias consultas (padre + hijos) y todas leen el mismo estado
3 **Consultas con resultados muy grandes**	`Junto a @Query`	Lectura coherente aunque se lea en varios bloques (CursorWindow) mientras otra parte escribe

¿Cuándo NO hace falta?

`@Insert` `@Update` `@Delete` `@Upsert` `@Query("UPDATE…")`

Ya se ejecutan siempre dentro de una transacción: añadirles @Transaction no tiene ningún efecto.

upsertConservandoFavorito: sin y con @Transaction

`@Transaction`

Sin @Transaction

`T1 · insertarIgnorando()` — COMMIT ✓

⚠ Excepción (p. ej. un valor nulo)

`T2, T3… · actualizarDesdeRed()` — No se ejecutan

Los libros nuevos ya están guardados, pero los existentes no se han actualizado.

✕ BD a medio sincronizar

Con @Transaction

UNA sola transacción

`insertarIgnorando()`

`actualizarDesdeRed()` × N

Excepción → ROLLBACK de todo

También se deshacen las inserciones del primer paso: la BD vuelve al estado inicial.

✓ BD siempre coherente

Reglas de uso de @Transaction

`@Transaction`

La función debe tener cuerpo

En un DAO interface se escribe con implementación; en un DAO abstract class, no puede ser abstract.

suspend o Flow

suspend: asíncrona, sin bloquear el hilo principal. Flow: la transacción se aplica en cada ejecución de la consulta.

Solo operaciones de BD

Nada de Retrofit ni esperas largas: las escrituras van de una en una y el resto espera en cola.

La excepción debe salir

Si se captura con try/catch dentro del cuerpo, Room da la función por buena y confirma los cambios.

Resumen

- Varias operaciones del DAO que deben aplicarse juntas
- Una @Query que devuelve una clase con @Relation
- Un @Insert, @Update o @Delete aislado: no es necesaria

Simulador: Escrituras, favoritos y @Transaction

@Database: una única puerta de entrada

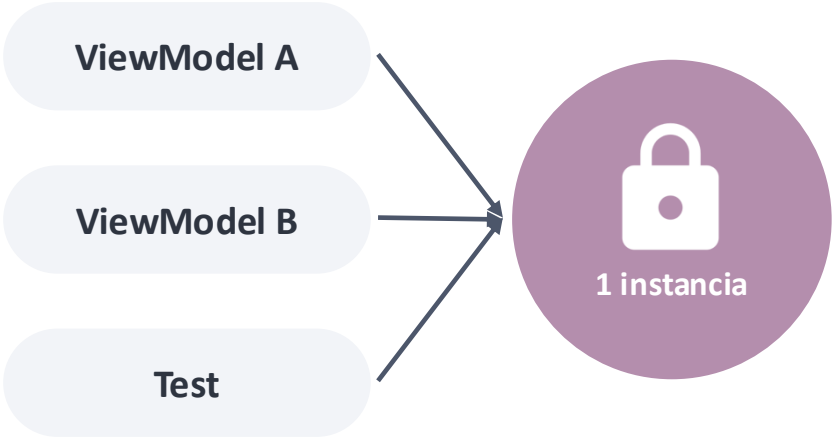
`@Database`

data/datasource/local/AppDatabase.kt

```kotlin
@Database(entities = [Libro::class],
    version = 1, exportSchema = true)
abstract class AppDatabase : RoomDatabase() {
    abstract fun libroDao(): LibrosDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context) =
            INSTANCE ?: synchronized(this) {
                Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "appdummy_database")
                .fallbackToDestructiveMigration(true)
                .build().also { INSTANCE = it }
            }
    }
}
```



`@Volatile` — El valor de INSTANCE es visible para todos los hilos.

`synchronized` — Evita que dos hilos creen la BD a la vez.

`applicationContext` — Evita retener una Activity en memoria.

Versiones y migraciones del esquema

`@Database`



Cada cambio de esquema exige incrementar version

`fallbackToDestructiveMigration`

Borra y recrea la BD si cambia el esquema. Válido en proyectos de aprendizaje: los datos se pierden.

`Objetos Migration`

Opción de producción: transforman el esquema conservando los datos. Requieren el esquema exportado en /schemas.

Vídeo del tema: Entidad, DAO y Database

03

Más allá de lo básico

TypeConverters y relaciones entre tablas



TypeConverters: tipos que SQLite no entiende

`@TypeConverter`

SQLite solo almacena:

`INTEGER` `REAL` `TEXT` `BLOB`

`List<String>` ⟷ `TEXT (JSON)`

```
Gson().toJson(value)
```

`java.util.Date` ⟷ `INTEGER (Long)`

```
date?.time
```

Dependencia:

```
implementation("com.google.code.gson:gson:2.14.0")
```

Converters.kt

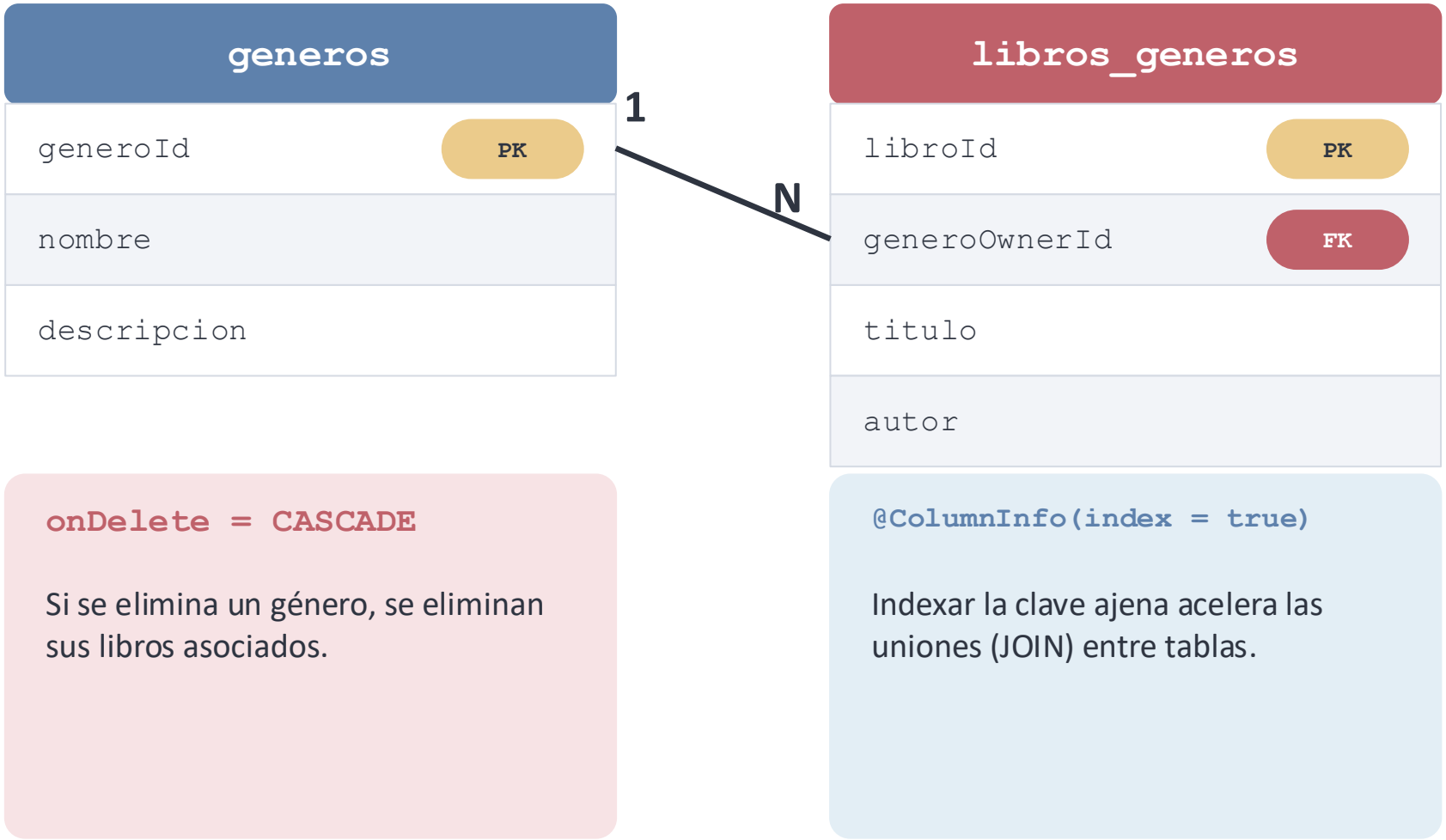
```
class Converters {
    @TypeConverter
    fun stringListToJson(value: List<String>) =
        Gson().toJson(value)

    @TypeConverter
    fun jsonToStringList(value: String):
        List<String> {
        val type = object :
            TypeToken<List<String>>() {}.type
        return Gson().fromJson(value, type)
    }
}

@Database(...)
@TypeConverters(Converters::class)
abstract class AppDatabase : RoomDatabase()
```

Relaciones 1:N entre tablas

`@ForeignKey`




```
@Entity(
 tableName =
  "libros_generos",
 foreignKeys = [
  ForeignKey(
   entity =
    Genero::class,
   parentColumns =
    ["generoId"],
   childColumns =
    ["generoOwnerId"],
   onDelete =
    ForeignKey.CASCADE)
 ])
```

`onDelete = CASCADE`

Si se elimina un género, se eliminan sus libros asociados.

`@ColumnInfo(index = true)`

Indexar la clave ajena acelera las uniones (JOIN) entre tablas.

Simulador: Relaciones y CASCADE

@Relation: combinar tablas en un resultado

@Relation

GeneroConLibros

NO es @Entity

@Embedded genero: Genero

Todas las columnas del padre

@Relation libros: List<LibroGenero>

```
parentColumn = "generoId"
entityColumn = "generoOwnerId"
```

@Transaction es imprescindible
Varias consultas internas con el mismo estado. Si falta, Room avisa al compilar.

GeneroDao.kt · GeneroViewModel.kt

```kotlin
@Dao
interface GeneroDao {
    @Transaction
    @Query(SELECT * FROM generos
           ORDER BY nombre ASC)
    fun observarGenerosConLibros():
        Flow<List<GeneroConLibros>>
}

// ViewModel
val generosConLibros = repository
    .observarGenerosConLibros()
    .stateIn(viewModelScope,
        SharingStarted.WhileSubscribed(5000),
        emptyList())
```

Vídeo del tema: Relaciones 1:N

04

Room en la arquitectura

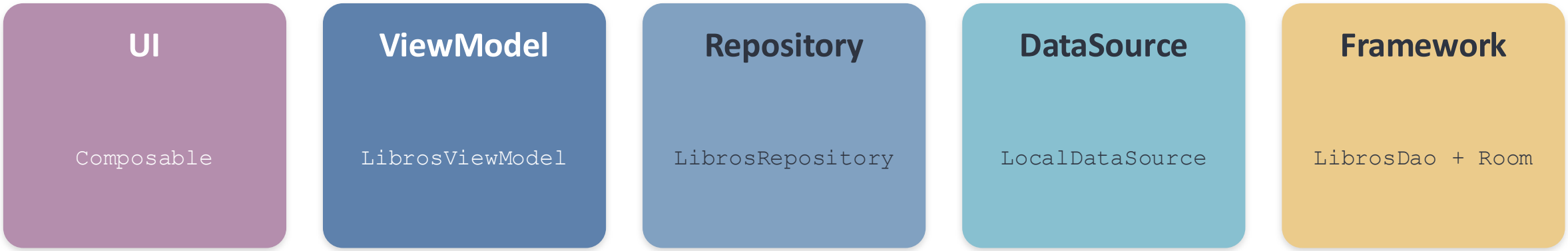
Repository, Flow frío y caliente, ViewModel y AppContainer



Flujo de capas con Room

MVVM

Datos observados: Flow<List<Libro>>



Acciones: toggleFavorito(id) — suspend

Regla de dependencias: cada capa solo conoce a la siguiente. El repositorio no sabe que existe Room; la UI no sabe que existe el repositorio.

LocalDataSource y LibrosRepository

Capa de datos

data/datasource/local/LocalDataSource.kt

```
class LocalDataSource(
    private val dao: LibrosDao
) {
    fun observarTodos(): Flow<List<Libro>> =
        dao.observarTodos()

    fun observarFavoritos(): Flow<List<Libro>> =
        dao.observarFavoritos()

    suspend fun toggleFavorito(id: String) =
        dao.toggleFavorito(id)
}
```

data/repository/LibrosRepository.kt

```
class LibrosRepository(
    private val localDataSource: LocalDataSource
) {
    // La UI siempre observa desde Room
    fun observarLibros(): Flow<List<Libro>> =
        localDataSource.observarTodos()

    suspend fun toggleFavorito(id: String) =
        localDataSource.toggleFavorito(id)

    // T6 → añadirá RemoteDataSource
}
```

Envuelve el DAO: aísla Room del resto de la app

Fuente única de verdad: lo que hay en Room

Flow frío y Flow caliente

Flow

Flow frío

Perezoso: una ejecución por cada colector

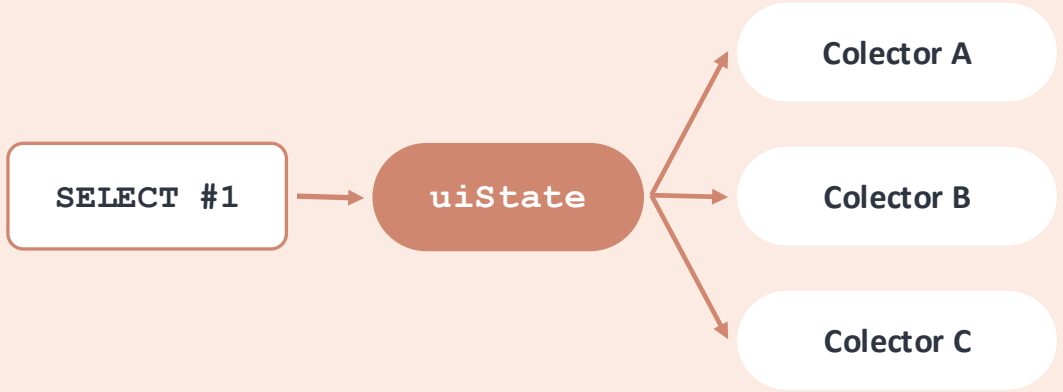


No se ejecuta hasta que alguien lo recolecta.

Ejemplos: `flow { }`, `flowOf()`, consultas Flow del DAO

Flow caliente

Compartido: una ejecución para todos



Existe aunque nadie lo recolecte y guarda su estado.

Ejemplos: `StateFlow`, `SharedFlow`, `stateIn`, `shareIn`

Diferencias entre Flow frío y caliente

`Flow`

Característica	Flow frío	Flow caliente (StateFlow / SharedFlow)
¿Cuándo produce datos?	Solo cuando se recolecta	Aunque no haya colectores (con stateIn, según SharingStarted)
Varios colectores	Cada uno lanza su propia ejecución	Todos comparten la misma ejecución
Valor actual	No tiene	StateFlow sí (.value); SharedFlow no
Un colector nuevo recibe…	La secuencia completa desde el principio	StateFlow: el último valor · SharedFlow: los replay últimos
Finalización	Al terminar su bloque (el Flow de Room no termina)	Nunca por sí solo: se cancela con su ámbito
Ejemplos	`flow { }, flowOf(), Flow del DAO`	`MutableStateFlow, MutableSharedFlow, stateIn, shareIn`

De frío a caliente: `stateIn → StateFlow · shareIn → SharedFlow`

Ejemplo: la misma consulta, fría y caliente

Flow

Ejemplo.kt (versión resumida)

```
fun main() = runBlocking {
    val consulta = flow {           // Flow frío
        println("▶ Consulta ejecutada")
        emit(listOf("Hail Mary", "Juego de tronos"))
    }
    println("Flow creado: no se ha ejecutado nada")

    // 1) FRÍO: dos colectores → dos ejecuciones
    consulta.collect { println("A recibe $it") }
    consulta.collect { println("B recibe $it") }

    // 2) CALIENTE: una ejecución compartida
    val ambito = CoroutineScope(Dispatchers.Default)
    val estado = consulta.stateIn(ambito)
    println("C lee ${estado.value}")
    estado.take(1).collect { println("D recibe $it") }
    ambito.cancel()   // como al destruirse el ViewModel
}
```

Consola (salida esperada)

```
Flow creado: no se ha ejecutado nada
▶ Consulta ejecutada
A recibe [Hail Mary, Juego de tronos]
▶ Consulta ejecutada
B recibe [Hail Mary, Juego de tronos]
▶ Consulta ejecutada
C lee [Hail Mary, Juego de tronos]
D recibe [Hail Mary, Juego de tronos]
```

- A y B: el flujo frío repite la consulta en cada collect
- 3.ª ejecución: la lanza stateIn, una sola vez
- C y D: leen el valor guardado, sin consultar

¿Por qué stateIn en el ViewModel?

`stateIn`

Si la UI recolectara el Flow frío del repositorio…

Una consulta por colector

Cada colector lanza su propio SELECT y su propio observador.

Sin valor actual

Cada colector de la UI tendría que indicar un valor inicial.

Se repite al volver

Tras una rotación, collectAsStateWithLifecycle vuelve a recolectar y consulta de nuevo.

… con stateIn

- Una única suscripción a Room en viewModelScope, compartida por todos
- Siempre hay valor: LibrosUiState.Cargando mientras llega el primero
- WhileSubscribed(5_000): la rotación no repite la consulta

Regla práctica



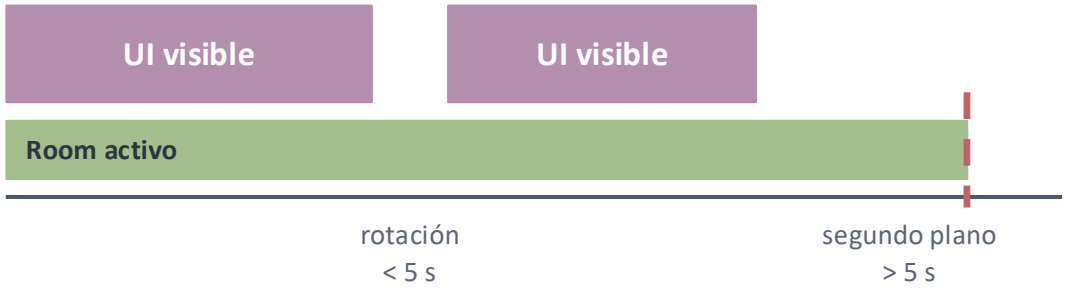
ViewModel: de Flow frío a StateFlow con stateIn

ViewModel

screens/listado/LibrosViewModel.kt

```
val uiState: StateFlow<LibrosUiState> =
    repository.observarLibros()
        .map<List<Libro>, LibrosUiState> {
            LibrosUiState.Exito(it)
        }
        .catch { e ->
            emit(LibrosUiState.Error(
                e.message ?: "Error"))
        }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted
                .WhileSubscribed(5000),
            initialValue = LibrosUiState.Cargando
        )
```

WhileSubscribed(5_000) en el tiempo



Flow frío — Solo produce datos cuando alguien lo recolecta.

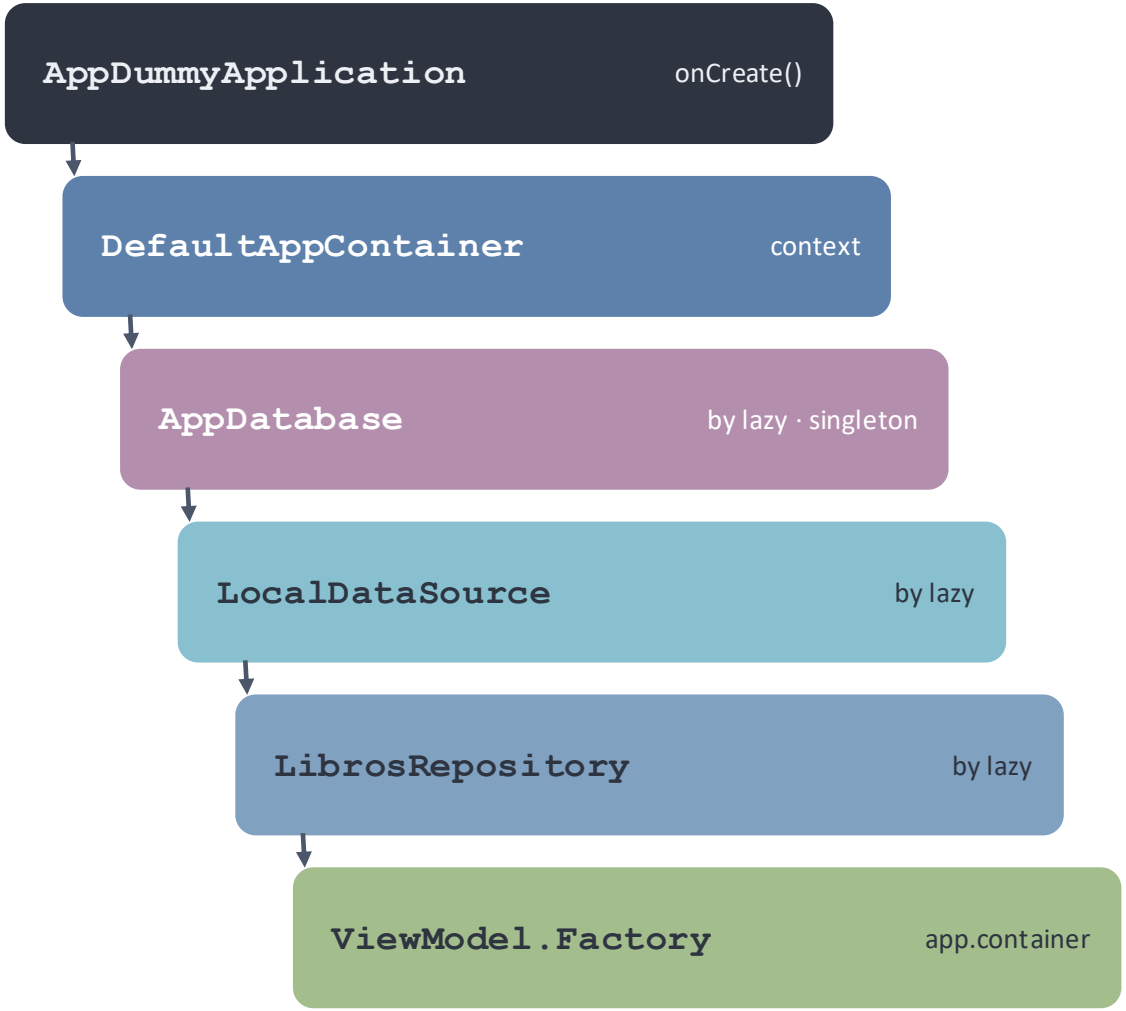
stateIn — Lo convierte en un StateFlow caliente con valor inicial.

5 000 ms — Sobrevive a rotaciones y libera recursos en segundo plano.

Vídeo del tema: Flujo reactivo con Room

AppContainer: dependencias sin Hilt ni Koin

DI manual



data/di/AppContainer.kt

```
class DefaultAppContainer(context: Context)
    : AppContainer {
    private val database by lazy {
        AppDatabase.getDatabase(context) }
    private val localDataSource by lazy {
        LocalDataSource(database.libroDao()) }
    override val librosRepository by lazy {
        LibrosRepository(localDataSource) }
}
```

AndroidManifest.xml

```
<application
    android:name=".AppDummyApplication" ...>
```

05

Probar y practicar

Testing con base de datos en memoria y desarrollo guiado



Testing de Room: base de datos en memoria

androidTest



Se crea en RAM y desaparece al terminar: no deja rastro en el dispositivo.

insertarYRecuperarLibro	Un libro nuevo se guarda con esFavorito = false
toggleFavoritaInvierteElEstado	false → true → false
upsertConservandoFavorito…	Actualiza el título y mantiene el favorito
observarFavoritasEmiteSolo…	first() del Flow devuelve solo los marcados

androidTest/LibroDaoTest.kt

```kotlin
@Before
fun crearBaseDeDatos() {
    database = Room.inMemoryDatabaseBuilder(
        ApplicationProvider
            .getApplicationContext(),
        AppDatabase::class.java)
        .allowMainThreadQueries() // solo tests
        .build()
    dao = database.libroDao()
}

@Test
fun toggleFavoritaInvierteElEstado() = runTest {
    dao.upsert(libro)
    dao.toggleFavorito("2")
    assertTrue(dao.obtenerPorId("2")!!.esFavorito)
}
```

Estructura de paquetes al final de T4

AppDummy

```
com.ejemplo.appdummy/
├─ AppDummyApplication.kt
├─ data/
│   ├─ model/Libro.kt
│   ├─ datasource/local/
│   │   ├─ AppDatabase.kt
│   │   ├─ Converters.kt
│   │   ├─ LibrosDao.kt
│   │   ├─ LocalDataSource.kt
│   ├─ repository/LibrosRepository.kt
│   ├─ di/AppContainer.kt
├─ screens/  (listado · detalle · favoritos)
├─ navegacion/
```

Nuevo en T4

Room, DataSource, contenedor

Modificado

Entidad, repositorio y pantallas

Sin cambios

Navegación

En T5, Libro.kt se ampliará con @SerializedName para Retrofit2.

Desarrollo práctico guiado: AppDummy con Room

Práctica

1 **Factories desde AppContainer**

Todos los ViewModel obtienen el repositorio del contenedor.

2 **Quitar «Reintentar»**

El botón de error no es necesario por ahora.

3 **getLibros() sin suspend**

Devuelve la lista estática para simular la carga.

4 **FloatingActionButton de carga**

Inserta la lista estática en Room (en T5, la API).

5 **Marcar como leído**

Igual que favoritos, sobre la columna leido.

6 **Detalle desde Room**

El ViewModel de detalle consulta el libro por id.

7 **Favoritos desde Room**

Solo los libros marcados como favoritos.



Punto de partida: release v5 de AppDummy (Ejemplo práctico Tema 3)

Ideas clave

KSP, no KAPT

Room genera el código de acceso al compilar.

Flow para observar

Sin suspend: Room re-emite al cambiar la tabla.

suspend para escribir

Insert, Update, Delete y lecturas puntuales.

Una sola AppDatabase

Singleton con @Volatile y synchronized.

Cuidado con REPLACE

Reescribe la fila y borra los datos locales.

Frío abajo, caliente arriba

DAO y Repository: Flow frío. El ViewModel lo calienta con stateIn.

Referencias

Fuentes

Guardar datos en una base de datos local con Room
developer.android.com/training/data-storage/room

Definir datos con entidades de Room
developer.android.com/training/data-storage/room/defining-data

Acceder a los datos con DAOs de Room
developer.android.com/training/data-storage/room/accessing-data

Relaciones entre objetos en Room
developer.android.com/training/data-storage/room/relationships

Versiones de Room
developer.android.com/jetpack/androidx/releases/room

Releases de KSP
github.com/google/ksp/releases

Tema 4. ROOM: Persistencia local (apuntes)
documentation.javiercarrasco.es/pmdm/B3/B3-T4_Room_Persistencia

StateFlow y SharedFlow — Android Developers
developer.android.com/kotlin/flow/stateflow-and-sharedflow

Flujos fríos (cold flows) — Documentación de Kotlin
kotlinlang.org/docs/coroutines-flow.html

Anexo B2-A1: StateFlow, corrutinas y arquitectura MVVM
documentation.javiercarrasco.es/pmdm/B2/B2-A1_Referencia_MVVM_StateFlow

Anexo B3-A1: Room avanzado, migraciones, testing y API Key segura
documentation.javiercarrasco.es/pmdm/B3/B3-A1_Referencia_Room_Retrofit